IRPF90: a programming environment for high performance computing

Anthony Scemama
Laboratoire de Chimie et Physique Quantiques,
CNRS-UMR 5626,
IRSAMC Université Paul Sabatier,
118 route de Narbonne
31062 Toulouse Cedex, France
(Dated: September 28, 2009)

Abstract

IRPF90 is a Fortran programming environment which helps the development of large Fortran codes. In Fortran programs, the programmer has to focus on the order of the instructions: before using a variable, the programmer has to be sure that it has already been computed in all possible situations. For large codes, it is common source of error. In IRPF90 most of the order of instructions is handled by the pre-processor, and an automatic mechanism guarantees that every entity is built before being used. This mechanism relies on the {needs/needed by} relations between the entities, which are built automatically. Codes written with IRPF90 execute often faster than Fortran programs, are faster to write and easier to maintain.

I. INTRODUCTION

The most popular programming languages in high performance computing (HPC) are those which produce fast executables (Fortran and C for instance). Large programs written in these languages are difficult to maintain and these languages are in constant evolution to facilitate the development of large codes. For example, the C++ language[1] was proposed as an improvement of the C language by introducing classes and other features of object-oriented programming. In this paper, we propose a Fortran pre-processor with a very limited number of keywords, which facilitates the development of large programs and the re-usability of the code without affecting the efficiency.

In the imperative programming paradigm, a computation is a ordered list of commands that change the state of the program. At the lowest level, the machine code is imperative: the commands are the machine code instructions and the state of the program is represented by to the content of the memory. At a higher level, the Fortran language is an imperative language. Each statement of a Fortran program modifies the state of the memory.

In the functional programming paradigm, a computation is the evaluation of a function. This function, to be evaluated, may need to evaluate other functions. The state of the program is not known by the programmer, and the memory management is handled by the compiler.

Imperative languages are easy to understand by machines, while functional languages are easy to understand by human beings. Hence, code written in an imperative language can be made extremely efficient, and this is the main reason why Fortran and C are so popular in the field of High Performance Computing (HPC).

However, codes written in imperative languages usually become excessively complicated to maintain and to debug. In a large code, it is often very difficult for the programmer to have a clear image of the state of the program at a given position of the code, especially when side-effects in a procedure modifiy memory locations which are used in other procedures.

In this paper, we present a tool called "Implicit Reference to Parameters with Fortran 90" (IRPF90). It is a Fortran pre-processor which facilitates the development of large simulation codes, by allowing the programmer to focus on *what* is being computed, instead of *how* it is computed. This last sentence often describes the difference between the functional and the imperative paradigms[2]. From a practical point of view, IRPF90 is a program written in the Python[3] language. It produces Fortran source files from IRPF90 source files. IRPF90 source files are Fortran source files with a limited number of additional statements. To explain how to use the IRPF90 tool, we will write a simple molecular dynamics program as a tutorial.

II. TUTORIAL: A MOLECULAR DYNAMICS PROGRAM

A. Imperative and functional implementation of the potential

We first choose to implement the Lennard-Jones potential[4] to compute the interaction of pairs of atoms:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{1}$$

```
 1  program potential_with_imperative_style
 2   implicit none
 3   double precision :: sigma_lj, epsilon_lj
 4   double precision :: interatomic_distance
 5   double precision :: sigma_over_r
 6   double precision :: V_lj
 7   print *, 'Sigma?'
 8   read(*,*) sigma_lj
 9   print *, 'Epsilon?'
10   read(*,*) epsilon_lj
11   print *, 'Interatomic Distance?'
12   read(*,*) interatomic_distance
13   sigma_over_r = sigma_lj/interatomic_distance
14   V_lj = 4.d0 * epsilon_lj * ( sigma_over_r**12 &
15    -  sigma_over_r**6 )
16   print *, 'Lennard-Jones potential:'
17   print *, V_lj
18  end program
```

FIG. 1: Imperative implementation of the Lennard-Jones potential.

where r is the atom-atom distance, ϵ is the depth of the potential well and σ is the value of r for which the potential energy is zero. ϵ and σ are the parameters of the force field.

Using an imperative style, one would obtain the program given in figure 1. One can clearly see the sequence of statements in this program: first read the data, then compute the value of the potential.

This program can be re-written using a functional style, as shown in figure 2. In the functional form of the program, the sequence of operations does not appear as clearly as in the imperative example. Moreover, the order of execution of the commands now depends on the choice of the compiler: the function `sigma_over_r` and the function `epsilon_lj` are both called on line 12-13, and the order of execution may differ from one compiler to the other.

The program was written in such a way that the functions have no arguments. The reason for this choice is that the references to the entities which are needed to calculate a function appear inside the function, and not outside of the function. Therefore, the code is simpler to understand for a programmer who never read this particular code, and it can be easily represented as a production tree (figure 3, above). This tree exhibits the relation {needs/needed by} between the entities of interest: the entity `V_lj` needs the entities `sigma_over_r` and `epsilon_lj` to be produced, and `sigma_over_r` needs `sigma_lj` and `interatomic_distance`.

In the imperative version of the code (figure 1), the production tree has to be known by the programmer so he can place the instructions in the proper order. For simple programs it is not a problem, but for large codes the production tree can be so large that the programmer is likely to make wrong assumptions in the dependencies between the entities. This complexifies the structure of the code by the introduction of many different methods to compute the same quantity, and the performance of the code can be reduced due to the computation of entities which are not needed.

In the functional version (figure 2), the production tree does not need to be known by the programmer. It exists implicitely through the function calls, and the evaluation of the main function is realized by exploring the tree with a depth-first algorithm. A large advantage of the functional style is that there can only be one way to calculate the value of an entity:

```
 1  program potential_with_functional_style
 2   double precision :: V_lj
 3   print *, V_lj()
 4  end program
 5
 6  double precision function V_lj()
 7   double precision :: sigma_lj
 8   double precision :: epsilon_lj
 9   double precision :: interatomic_distance
10   double precision :: sigma_over_r
11   V_lj = 4.d0 * epsilon_lj() * &
12     ( sigma_over_r()**12 - sigma_over_r()**6 )
13  end function
14
15  double precision function epsilon_lj()
16   print *, 'Epsilon?'
17   read(*,*) epsilon_lj
18  end function
19
20  double precision function sigma_lj ()
21   print *, 'Sigma?'
22   read(*,*) sigma_lj
23  end function
24
25  double precision function sigma_over_r()
26   double precision :: sigma_lj
27   double precision :: interatomic_distance
28   sigma_over_r = sigma_lj()/interatomic_distance()
29  end function
30
31  double precision function interatomic_distance()
32   print *, 'Interatomic Distance?'
33   read(*,*) interatomic_distance
34  end function
```

FIG. 2: Functional implementation of the Lennard-Jones potential.

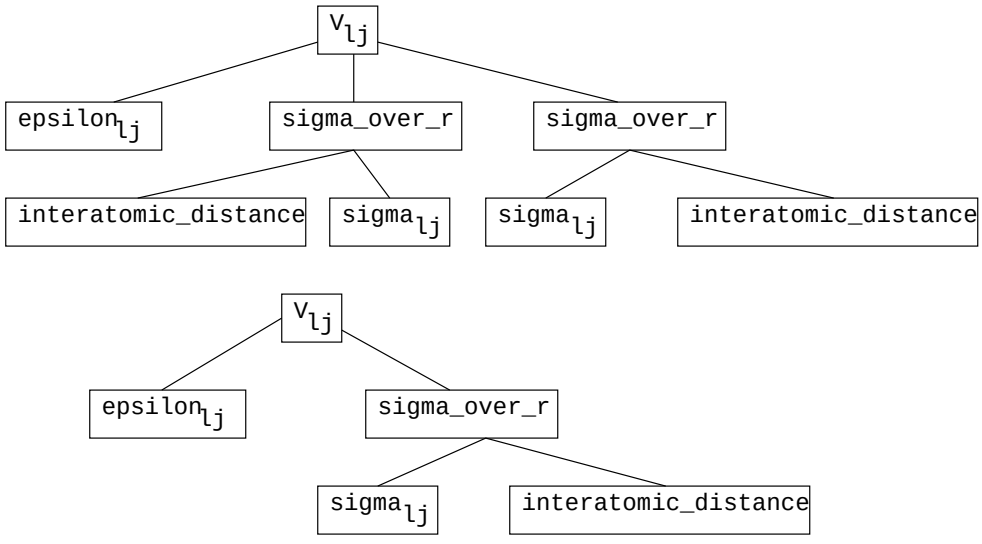

FIG. 3: The production tree of `V_lj`. Above, the tree produced by the program of figure 2. Below, the tree obtained if only one call to `sigma_over_r` is made.

```
 1  double precision function sigma_over_r()
 2   double precision       :: sigma_lj
 3   double precision       :: interatomic_distance
 4   double precision, save :: last_result
 5   integer, save          :: first_time_here
 6   if (first_time_here.eq.0) then
 7     last_result = sigma_lj()/interatomic_distance()
 8     first_time_here = 1
 9   endif
10   sigma_over_r = last_result
11  end function
```

FIG. 4: Memoized `sigma_over_r function`

calling the corresponding function. Therefore, the readability of the code is improved for a programmer who is not familiar with the program. Moreover, as soon as an entity is needed, it is calculated and valid. Writing programs in this way reduces considerably the risk to use un-initialized variables, or variables that are supposed to have a given value but which have been modified by a side-effect.

With the functional example, every time a quantity is needed it is computed, even if it has already been built before. If the functions are pure (with no side-effects), one can implement memoization[5, 6] to reduce the computational cost: the last value of the function is saved, and if the function is called again with the same arguments the last result is returned instead of computing it again. In the present example we chose to write functions with no arguments, so memoization is trivial to implement (figure 4). If we consider that the leaves of the production tree are constant, memoization can be applied to all the functions. The production tree of `V_lj` can now be simplified, as shown in figure 3, below.

B. Presentation of the IRPF90 statements

IRPF90 is a Fortran pre-processor: it generates Fortran code from source files which contain keywords specific to the IRPF90 program. The keywords understood by IRPF90 pre-processor are briefly presented. They will be examplified in the next subsections for the molecular dynamics example.

`BEGIN_PROVIDER ... END_PROVIDER`
Delimitates the definition of a provider (sections II C and II D).

`BEGIN_DOC ... END_DOC`
Delimitates the documentation of the current provider (section II C).

`BEGIN_SHELL ... END_SHELL`
Delimitates an embedded script (section II E).

`ASSERT`
Expresses an assertion (section II C).

`TOUCH`
Expresses the modification of the value of an entity by a side-effect (section II F).

`FREE`
Invalidates an entity and free the associated memory. (section **??**).

`IRP_READ / IRP_WRITE`
Reads/Writes the content of the production tree to/from disk (section II G).

```
program lennard_jones_dynamics
 print *, V_lj
end program

BEGIN_PROVIDER [ double precision, V_lj ]
  implicit none
  BEGIN_DOC
! Lennard Jones potential energy.
  END_DOC
  double precision :: sigma_over_r
  sigma_over_r = sigma_lj / interatomic_distance
  V_lj = 4.d0 * epsilon_lj * ( sigma_over_r**12 &
     - sigma_over_r**6 )
END_PROVIDER

 BEGIN_PROVIDER [ double precision, epsilon_lj ]
&BEGIN_PROVIDER [ double precision, sigma_lj ]
  BEGIN_DOC
! Parameters of the Lennard-Jones potential
  END_DOC
  print *, 'Epsilon?'
  read(*,*) epsilon_lj
  ASSERT (epsilon_lj > 0.)
  print *, 'Sigma?'
  read(*,*) sigma_lj
  ASSERT (sigma_lj > 0.)
END_PROVIDER

BEGIN_PROVIDER[double precision,interatomic_distance]
  BEGIN_DOC
! Distance between the atoms
  END_DOC
  print *, 'Inter-atomic distance?'
  read (*,*) interatomic_distance
  ASSERT (interatomic_distance >= 0.)
END_PROVIDER
```

FIG. 5: IRPF90 implementation of the Lennard-Jones potential.

`IRP_IF ... IRP_ELSE ... IRP_ENDIF`
Delimitates blocks for conditional compilation (section II G).
`PROVIDE`
Explicit call to the provider of an entity (section II G).

C. Implementation of the potential using IRPF90

In the IRPF90 environment, the entities of interest are the result of memoized functions with no arguments. This representation of the data allows its organization in a production tree, which is built and handled by the IRPF90 pre-processor. The previous program may be written again using the IRPF90 environment, as shown in figure 5.

The program shown in figure 5 is very similar to the functional program of figure 2. The difference is that the entities of interest are not functions anymore, but variables. The variable corresponding to an entity is provided by calling a providing procedure (or provider), defined between the keywords `BEGIN_PROVIDER ... END_PROVIDER`. In the IRPF90 environment, a provider can provide several entities (as shown with the parameters of the potential), although it is preferable to have providers that provide only one entity.

When an entity has been built, it is tagged as built. Hence, the next call to the provider will return the last computed value, and will not build the value again. This explains why in the IRPF90 environment the parameters of the force field are asked only once to the user.

The `ASSERT` keyword was introduced to allow the user to place assertions[9] in the code. An assertion specifies certain general properties of a value. It is expressed as a logical expression which is supposed to be always true. If it is not, the program is wrong. Assertions in the code provide run-time checks which can dramatically reduce the time spent finding bugs: if an assertion is not verified, the program stops with a message telling the user which assertion caused the program to fail.

The `BEGIN_DOC ... END_DOC` blocks contain the documentation of the provided entities. The descriptions are encapsulated inside these blocks in order to facilitate the generation of technical documentation. For each entity a "man page" is created, which contains the {needs/needed by} dependencies of the entity and the description given in the `BEGIN_DOC ... END_DOC` block. This documentation can be accessed by using the `irpman` command followed by the name of the entity.

The IRPF90 environment was created to simplify the work of the scientific programmer. A lot of time is spent creating Makefiles, which describe the dependencies between the source files for the compilation. As the IRPF90 tool "knows" the production tree, it can build automatically the Makefiles of programs, without any interaction with the user. When the user starts a project, he runs the command `irpf90 -init` in an empty directory. A standard Makefile is created, with the gfortran compiler[10] as a default. Then, the user starts to write IRPF90 files which contain providers, subroutines, functions and main programs in files characterized by the `.irp.f` suffix. Running `make` calls `irpf90`, and a correct Makefile is automatically produced and used to compile the code.

D. Providing arrays

Now the basics of IRPF90 are known to the reader, we can show how simple it is to write a molecular dynamics program. As we will compute the interaction of several atoms, we will change the previous program such that we produce an array of potential energies per atom. We first need to introduce the quantity `Natoms` which contains the number of atoms. Figure 6 shows the code which defines the geometrical parameters of the system. Figure 7 shows the providers corresponding to the potential energy V per atom i, where it is chosen equal to the Lennard-Jones potential energy:

$$V_i = V_i^{LJ} = \sum_{j \neq i}^{N_{\text{atoms}}} 4\epsilon \left[\left(\frac{\sigma}{||\mathbf{r}_{ij}||} \right)^{12} - \left(\frac{\sigma}{||\mathbf{r}_{ij}||} \right)^{6} \right] \tag{2}$$

Figure 8 shows the providers corresponding to the kinetic energy T per atom i:

$$T_i = \frac{1}{2} m_i ||\mathbf{v}_i||^2 \tag{3}$$

where m_i is the mass and $\mathbf{v}_i$ is the velocity vector of atom i. The velocity vector is chosen to be initialized zero.

The dimensions of arrays are given in the definition of the provider. If an entity, defines the dimension of an array, the provider of the dimensioning entity will be called before allocating the array. This guarantees that the array will always be allocated with the proper

```
BEGIN_PROVIDER [ integer, Natoms ]
 BEGIN_DOC
! Number of atoms
 END_DOC
 print *, 'Number of atoms?'
 read(*,*) Natoms
 ASSERT (Natoms > 0)
END_PROVIDER

 BEGIN_PROVIDER [ double precision, coord, (3,Natoms) ]
&BEGIN_PROVIDER [ double precision, mass , (Natoms)   ]
 implicit none
 BEGIN_DOC
! Atomic data, input in atomic units.
 END_DOC
 integer :: i,j
 print *, 'For each atom: x, y, z, mass?'
 do i=1,Natoms
  read(*,*) (coord(j,i), j=1,3), mass(i)
  ASSERT (mass(i) > 0.)
 enddo
END_PROVIDER

BEGIN_PROVIDER[double precision,distance,(Natoms,Natoms)]
  implicit none
  BEGIN_DOC
! distance  : Distance matrix of the atoms
  END_DOC
  integer :: i,j,k
  do i=1,Natoms
   do j=1,Natoms
    distance(j,i) = 0.
    do k=1,3
     distance(j,i) = distance(j,i) + &
       (coord(k,i)-coord(k,j))**2
    enddo
    distance(j,i) = sqrt(distance(j,i))
   enddo
  enddo
END_PROVIDER
```

FIG. 6: Code defining the geometrical parameters of the system

size. In IRPF90, the memory allocation of an array entity is not written by the user, but by the pre-processor.

Memory can be explicitely freed using the keyword `FREE`. For example, de-allocating the array `velocity` would be done using `FREE velocity`. If the memory of an entity is freed, the entity is tagged as "not built", and it will be allocated and built again the next time it is needed.

E. Embedding scripts

The IRPF90 environment allows the programmer to write scripts inside his code. The scripting language that will interpret the script is given in brackets. The result of the shell script will be inserted in the file, and then will be interpreted by the Fortran pre-processor. Such scripts can be used to write templates, or to write in the code some information that has to be retrieved at compilation. For example, the date when the code was compiled can

```
BEGIN_PROVIDER [ double precision, V, (Natoms) ]
  BEGIN_DOC
! Potential energy.
  END_DOC
  integer :: i
  do i=1,Natoms
   V(i) = V_lj(i)
  enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_lj, (Natoms) ]
  implicit none
  BEGIN_DOC
! Lennard Jones potential energy.
  END_DOC
  integer :: i,j
  double precision :: sigma_over_r
  do i=1,Natoms
   V_lj(i) = 0.
   do j=1,Natoms
    if ( i /= j ) then
      ASSERT (distance(j,i) > 0.)
      sigma_over_r = sigma_lj / distance(j,i)
      V_lj(i) = V_lj(i) + sigma_over_r**12 &
       - sigma_over_r**6
    endif
   enddo
   V_lj(i) = 4.d0 * epsilon_lj * V_lj(i)
  enddo
END_PROVIDER

 BEGIN_PROVIDER [ double precision, epsilon_lj ]
&BEGIN_PROVIDER [ double precision, sigma_lj ]
  BEGIN_DOC
! Parameters of the Lennard-Jones potential
  END_DOC
  print *, 'Epsilon?'
  read(*,*) epsilon_lj
  ASSERT (epsilon_lj > 0.)
  print *, 'Sigma?'
  read(*,*) sigma_lj
  ASSERT (sigma_lj > 0.)
END_PROVIDER
```

FIG. 7: Definition of the potential.

be inserted in the source code using the example given in figure 9.

In our molecular dynamics program, the total kinetic energy `E_kin` is the sum over all the elements of the kinetic energy vector `T`:

$$E_{\text{kin}} = \sum_{i=1}^{N_{\text{atoms}}} T_i \tag{4}$$

Similarly, the potential energy `E_pot` is the sum of all the potential energies per atom.

$$E_{\text{pot}} = \sum_{i=1}^{N_{\text{atoms}}} V_i \tag{5}$$

The code to build `E_kin` and `E_pot` is very close: only the names of the variables change, and it is convenient to write the code using a unique template for both quantities, as shown in

```
BEGIN_PROVIDER [ double precision, T, (Natoms) ]
 BEGIN_DOC
! Kinetic energy per atom
 END_DOC
 integer :: i
 do i=1,Natoms
  T(i) = 0.5d0 * mass(i) * velocity2(i)
 enddo
END_PROVIDER

BEGIN_PROVIDER[double precision,velocity2,(Natoms)]
 BEGIN_DOC
! Square of the norm of the velocity per atom
 END_DOC
 integer :: i, k
 do i=1,Natoms
  velocity2(i) = 0.d0
  do k=1,3
   velocity2(i) = velocity2(i) + velocity(k,i)**2
  enddo
 enddo
END_PROVIDER

BEGIN_PROVIDER[double precision,velocity,(3,Natoms)]
 BEGIN_DOC
! Velocity vector per atom
 END_DOC
 integer :: i, k
 do i=1,Natoms
  do k=1,3
   velocity(k,i) = 0.d0
  enddo
 enddo
END_PROVIDER
```

FIG. 8: Definition of the kinetic energy.

```
program print_the_date
 BEGIN_SHELL [ /bin/sh ]
  echo print *, \'Compiled by $USER on `date`\'
 END_SHELL
end program
```

FIG. 9: Embedded shell script which gets the date of compilation.

figure 10. In this way, adding a new property which is the sum over all the atomic properties can done be done in only one line of code: adding the triplet (Property, Documentation, Atomic Property) to the list of entities at line 15.

F. Changing the value of an entity by a controlled side-effect

Many computer simulation programs contain iterative processes. In an iterative process, the same function has to be calculated at each step, but with different arguments. In our IRPF90 environment, at every iteration the production tree is the same, but the values of some entities change. To keep the program correct, if the value of one entity is changed it has to be tagged as "built" with its new value, and all the entities which depend on this

```
BEGIN_SHELL [ /usr/bin/python ]
template = """
BEGIN_PROVIDER [ double precision, %(entity)s ]
 BEGIN_DOC
! %(doc)s
 END_DOC
 integer :: i
 %(entity)s = 0.
 do i=1,Natoms
  %(entity)s = %(entity)s+%(e_array)s(i)
 enddo
END_PROVIDER
"""
entities = [ ("E_pot", "Potential Energy", "V"),
             ("E_kin", "Kinetic Energy", "T") ]
for e in entities:
  dictionary = { "entity": e[0],
    "doc": e[1],
    "e_array": e[2]}
  print template%dictionary
END_SHELL
```

FIG. 10: Providers of the Lennard-Jones potential energy and the kinetic energy using a template.

entity (directly or indirectly) need to be tagged as "not built". These last entities will need to be re-computed during the new iteration. This mechanism is achieved automatically by the IRPF90 pre-processor using the keyword `TOUCH`. The side-effect modifying the value of the entity is controlled, and the program will stay consistent with the change everywhere in the rest of the code.

In our program, we are now able to compute the kinetic and potential energy of the system. The next step is now to implement the dynamics. We choose to use the velocity Verlet algorithm[11]:

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^n \Delta t + \mathbf{a}^n \frac{\Delta t^2}{2} \tag{6}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \frac{1}{2}(\mathbf{a}^n + \mathbf{a}^{n+1})\Delta t \tag{7}$$

where $\mathbf{r}^n$ and $\mathbf{v}^n$ are respectively the position and velocity vectors at step n, Δt is the time step and the acceleration vector $\mathbf{a}$ is defined as

$$\mathbf{a} = \sum_{i=1}^{N_{\text{atoms}}} -\frac{1}{m_i} \nabla_i E_{\text{pot}} \tag{8}$$

The velocity Verlet algorithm is written in a subroutine `verlet`, and the gradient of the potential energy ∇E_{pot} can be computed by finite difference (figure 11).

Computing a component i of the numerical gradient of E_{pot} can be decomposed in six steps:

1. Change the component i of the coordinate $\mathbf{r}_i \longrightarrow (\mathbf{r}_i + \delta)$
2. Compute the value of E_{pot}
3. Change the coordinate $(\mathbf{r}_i + \delta) \longrightarrow (\mathbf{r}_i - \delta)$

```
 1  BEGIN_PROVIDER  [ double precision, dstep ]
 2   BEGIN_DOC
 3  ! Finite difference step
 4   END_DOC
 5   dstep = 1.d-4
 6  END_PROVIDER
 7
 8  BEGIN_PROVIDER[double precision,V_grad_numeric,(3,Natoms)]
 9   implicit none
10   BEGIN_DOC
11  ! Numerical gradient of the potential
12   END_DOC
13   integer :: i, k
14   do i=1,Natoms
15    do k=1,3
16     coord(k,i) = coord(k,i) + dstep
17     TOUCH coord
18     V_grad_numeric(k,i) = E_pot
19     coord(k,i) = coord(k,i) - 2.d0*dstep
20     TOUCH coord
21     V_grad_numeric(k,i) = &
22       ( V_grad_numeric(k,i)-E_pot )/(2.d0*dstep)
23     coord(k,i) = coord(k,i) + dstep
24    enddo
25   enddo
26   TOUCH coord
27  END_PROVIDER
28
29  BEGIN_PROVIDER [ double precision, V_grad, (3,Natoms) ]
30   BEGIN_DOC
31  ! Gradient of the potential
32   END_DOC
33   integer :: i,k
34   do i=1,Natoms
35    do k=1,3
36     V_grad(k,i) = V_grad_numeric(k,i)
37    enddo
38   enddo
39  END_PROVIDER
```

FIG. 11: Provider of the gradient of the potential.

4. Compute the value of $E_{\rm pot}$

5. Compute the component of the gradient using the two last values of $E_{\rm pot}$

6. Re-set $(\mathbf{r}_i - \delta) \longrightarrow \mathbf{r}_i$

The provider of `V_grad_numeric` follows these steps: in the internal loop, the array `coord` is changed (line 16). Touching it (line 17) invalidates automatically `E_pot`, since it depends indirectly on `coord`. As the value of `E_pot` is needed in line 18 and not valid, it is re-computed between line 17 and line 18. The value of `E_pot` which is affected to `V_grad_numeric(k,i)` is the value of the potential energy, consistent with the current set of atomic coordinates. Then, the coordinates are changed again (line 19), and the program is informed of this change at line 20. When the value of `E_pot` is used again at line 22, it is consistent with the last change of coordinates. At line 23 the coordinates are changed again, but no touch statement follows. The reason for this choice is efficiency, since two cases are possible for the next instruction: if we are at the last iteration of the loop, we exit the main loop and

line 26 is executed. Otherwise, the next instruction will be line 16. Touching `coord` is not necessary between line 23 and line 16 since no other entity is used.

The important point is that the programmer doesn't have to know *how* `E_pot` depends on `coord`. He only has to apply a simple rule which states that when the value of an entity A is modified, it has to be touched before any other entity B is used. If B depends on A, it will be re-computed, otherwise it will not, and the code will always be correct. Using this method to compute a numerical gradient allows a programmer who is not familiar with the code to compute the gradient of any entity A with respect to any other quantity B, without even knowing if A depends on B. If A does not depend on B, the gradient will automatically be zero. In the programs dealing with optimization problems, it is a real advantage: a short script can be written to build automatically all the possible numerical derivatives, involving all the entities of the program, as given in figure 12.

The velocity Verlet algorithm can be implemented (figure 13) as follows:

1. Compute the new value of the coordinates
2. Compute the component of the velocities which depends on the old set of coordinates
3. Touch the coordinates and the velocities
4. Increment the velocities by their component which depends on the new set of coordinates
5. Touch the velocities

G. Other Features

As IRPF90 is designed for HPC, conditional compilation is an essential requirement. Indeed, it is often used for activating and deactivating blocks of code defining the behavior of the program under a parallel environment. This is achieved by the `IRP_IF`...`IRP_ELSE`...`IRP_ENDIF` constructs. In figure 14, the checkpointing block is activated by running `irpf90 -DCHECKPOINT`. If the `-D` option is not present, the other block is activated.

The current state of the production tree can written to disk using the command `IRP_WRITE` as in figure 14. For each entity in the subtrees of `E_pot` and `E_kin`, a file is created with the name of the entity which contains the value of the entity. The subtree can be loaded again later using the `IRP_READ` statement. This functionality is particularly useful for adding quickly a checkpointing feature to an existing program.

The `PROVIDE` keyword was added to assign imperatively a needs/needed by relation between two entities. This keyword can be used to associate the value of an entity to an iteration number in an iterative process, or to help the preprocessor to produce more efficient code.

A last convenient feature was added: the declarations of the local variables do not need anymore to be located before the first executable statement. The local variables can now be declared anywhere inside the providers, subroutines and functions. The IRPF90 preprocessor will put them at the beginning of the subroutines or functions for the programmer. It allows the user to declare the variables where the reader needs to know to what they correspond.

```
BEGIN_SHELL [ /usr/bin/python ]
# Read the names of the entities and their dimensions
dims = {}
import os
for filename in os.listdir('.'):
  if filename.endswith('.irp.f'):
    file = open(filename,'r')
    for line in file:
     if "%" not in line:
      if line.strip().lower().startswith('begin_provider'):
        buffer = line.split(',',2)
        name = buffer[1].split(']')[0].strip()
        if len(buffer) == 2:
          dims[name] = []
        else:
          dims[name] = buffer[2]
          for c in "()] \n":
            dims[name] = dims[name].replace(c,"")
          dims[name] = dims[name].split(",")
    file.close()
# The template to use for the code generation
template = """
BEGIN_PROVIDER[double precision, grad_%(var1)s_%(var2)s %(dims2)s]
 BEGIN_DOC
! Gradient of %(var1)s with respect to %(var2)s
 END_DOC
 integer :: %(all_i)s
 double precision :: two_dstep
 two_dstep = dstep + dstep
%(do)s
   %(var2)s %(indice)s =  %(var2)s %(indice)s + dstep
   TOUCH %(var2)s
   grad_%(var1)s_%(var2)s %(indice)s = %(var1)s
   %(var2)s %(indice)s =  %(var2)s %(indice)s - two_dstep
   TOUCH %(var2)s
   grad_%(var1)s_%(var2)s %(indice)s =  &
      (grad_%(var1)s_%(var2)s %(indice)s - %(var1)s)/two_dstep
   %(var2)s %(indice)s =  %(var2)s %(indice)s + dstep
%(enddo)s
 TOUCH %(var2)s
END_PROVIDER
"""
# Generate all possibilities of d(v1)/d(v2), with v1 scalar
for v1 in dims.keys():
 if dims[v1] == []:
  for v2 in dims.keys():
   if v2 != v1:
    do = ""
    enddo = ""
    if dims[v2] == []:
      dims2 = ""
      all_i = "i"
      indice = ""
    else:
      dims2 = ', ('+','.join(dims[v2])+')'
      all_i = ','.join([ "i"+str(k) for k in range(len(dims[v2]))])
      indice = "("
      for k,d in enumerate(dims[v2]):
        i = "i"+str(k)
        do = " do "+i+" = 1,"+d+"\n"+do
        enddo += " enddo\n"
        indice += i+","
      indice = indice[:-1]+")"
    dictionary = {"var1"  : v1,
     "var2"  : v2, "dims2" : dims2,
     "all_i" : all_i, "do"    : do,
     "indice": indice, "enddo" : enddo}
    print template%dictionary
END_SHELL
```


FIG. 12. Automatic generation of all possible gradients of scalar entities with respect to all other

```
 1  BEGIN_PROVIDER [ integer, Nsteps ]
 2   BEGIN_DOC
 3  ! Number of steps for the dynamics
 4   END_DOC
 5   print *, 'Nsteps?'
 6   read(*,*) Nsteps
 7   ASSERT (Nsteps > 0)
 8  END_PROVIDER
 9
10   BEGIN_PROVIDER [ double precision, tstep ]
11  &BEGIN_PROVIDER [ double precision, tstep2 ]
12   BEGIN_DOC
13  ! Time step for the dynamics
14   END_DOC
15   print *, 'Time step?'
16   read(*,*) tstep
17   ASSERT (tstep > 0.)
18   tstep2 = tstep*tstep
19  END_PROVIDER
20
21  BEGIN_PROVIDER[double precision,acceleration,(3,Natoms)]
22   implicit none
23   BEGIN_DOC
24  ! Acceleration = - grad(V)/m
25   END_DOC
26   integer :: i, k
27   do i=1,Natoms
28    do k=1,3
29     acceleration(k,i) = - V_grad(k,i)/mass(i)
30    enddo
31   enddo
32  END_PROVIDER
33
34  subroutine verlet
35   implicit none
36   integer :: is, i, k
37   do is=1,Nsteps
38    do i=1,Natoms
39     do k=1,3
40      coord(k,i) = coord(k,i) + tstep*velocity(k,i) + &
41       0.5*tstep2*acceleration(k,i)
42      velocity(k,i) = velocity(k,i) + 0.5*tstep* &
43       acceleration(k,i)
44     enddo
45    enddo
46    TOUCH coord velocity
47    do i=1,Natoms
48     do k=1,3
49      velocity(k,i) = velocity(k,i) + 0.5*tstep* &
50       acceleration(k,i)
51     enddo
52    enddo
53    TOUCH velocity
54    call print_data(is)
55   enddo
56  end subroutine
```

FIG. 13: The velocity Verlet algorithm.

```
1  program dynamics
2
3    call verlet
4
5  IRP_IF CHECKPOINT
6
7    print *, 'Checkpoint'
8    IRP_WRITE E_pot
9    IRP_WRITE E_kin
10
11 IRP_ELSE
12
13   print *, 'No checkpoint'
14
15 IRP_ENDIF
16
17 end
```

FIG. 14: The main program.

III. EFFICIENCY OF THE GENERATED CODE

In the laboratory, we are currently re-writing a quantum Monte Carlo (QMC) program, named QMC=Chem, with the IRPF90 tool. The same computation was realized with the old code (usual Fortran code), and the new code (IRPF90 code). Both codes were compiled with the Intel Fortran compiler version 11.1 using the same options. A benchmark was realized on an Intel Xeon 5140 processor.

The IRPF90 code is faster than the old code by a factor of 1.60: the CPU time of the IRPF90 executable is 62% of the CPU time of the old code. This time reduction is mainly due to the avoidance of computing quantities that are already computed. The total number of processor instructions is therefore reduced.

The average number of instructions per processor cycle is 1.47 for the old code, and 1.81 for the IRPF90 code. This application shows that even if the un-necessary computations were removed from the old code, the code produced by IRPF90 would still be more efficient. The reason is that in IRPF90, the programmer is guided to write efficient code: the providers are small subroutines that manipulate a very limited number of memory locations. This coding style improves the temporal locality of the code[12] and thus minimizes the number of cache misses.

The conclusion of this real-size application is that the overhead due to the management of the production tree is negligible compared to the efficiency gained by avoiding to compute many times the same quantity, and by helping the Fortran compiler to produce optimized code.

IV. SUMMARY

The IRPF90 environment is proposed for writing programs with reduced complexity. This technique for writing programs, called "Implicit Reference to Parameters" (IRP),[7] is conform to the recommendations of the "Open Structure Interfaceable Programming Environment" (OSIPE)[8]:

- Open: Unambiguous identification and access to any entity anywhere in the program

- Interfaceable: Easy addition of any new feature to an existing code
- Structured: The additions will have no effect on the program logic

The programming paradigm uses some ideas of functional programming and thus clarifies the correspondance between the mathematical formulas and the code. Therefore, scientists do not need to be experts in programming to write clear, reusable and efficient code, as shown with the simple molecular dynamics code presented in this paper.

The consequences of the locality of the code are multiple:

- the code is efficient since the temporal locality is increased,
- the overlap of pieces of code written simultaneously by multiple developers is reduced.
- regression testing[13] can be achieved by writing, for each entity, a program which tests that the entity is built correctly.

Finally, let us mention that the IRPF90 pre-processor generates Fortran 90 which is fully compatible with standard subroutines and functions. Therefore the produced Fortran code can be compiled on any architecture, and the usual HPC libraries (BLAS[14], LAPACK[15], MPI[16],...) can be used.

The IRPF90 program can be downloaded on http://irpf90.sourceforge.net

Acknowledgments

The author would like to acknowledge F. Colonna (CNRS, Paris) for teaching him the IRP method, and long discussions around this subject. The author also would like to thank P. Reinhardt (Université Pierre et Marie Curie, Paris) for testing and enjoying the IRPF90 tool, and F. Spiegelman (Université Paul Sabatier, Toulouse) for discussions about the molecular dynamics code.

---

[1] Stroustrup B. *The C++ Programming Language* Ed: Addison-Wesley Pub Co; 3rd edition (2000).
[2] Hudak P. *ACM Comput. Surv.* **21(3)**, 359 (1989).
[3] http://www.python.org/
[4] Lennard-Jones J. E., *Proceedings of the Physical Society* **43**, 461 (1931).
[5] Michie D. *Nature* **218** 19 (1968).
[6] Hughes R.J.M. "Lazy memo functions" in: G. Goos and J. Hartmanis, eds., Proc. Conf: on Functional Programming and Computer Architecture, Nancy, France, September 1985, Springer Lecture Note Series, Vol. 201 (Springer, Berlin, 1985).
[7] http://galileo.lct.jussieu.fr/~frames/mediawiki/index.php/IRP_Programming_Presentation
[8] Colonna F., Jolly L.-H., Poirier R. A., Ángyán J. G., and Jansen G. *Comp. Phys. Comm.* **81(3)**, 293 (1994).
[9] Hoare C.A.R., *Commun. ACM*, **12(10)**, 576 (1969).
[10] http://gcc.gnu.org/fortran/
[11] Swope W. C., Andersen H. C., Berens P. H., and Wilson K. R. *J. Chem. Phys.* **76**, 637 (1982).

[12] Denning P. J. *Commun. ACM* **48(7)**, 19 (2005).
[13] Agrawal H., Horgan J. R., Krauser, E.W., London, S., Incremental regression testing. in: Proceedings of the IEEE Conference on Software Maintenance, 348 (1993).
[14] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, *ACM Trans. Math. Soft.* **28(2)**, 135 (2002).
[15] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A. and Sorensen D. *LAPACK Users' Guide*, Ed: Society for Industrial and Applied Mathematics, Philadelphia, PA, (1999).
[16] Gropp W., Lusk E., Doss N. and A. Skjellum, *Parallel Computing* **22(6)**, 789 (1996).